MTI #1737

MODIFIED HARVARD ARCHITECTURE PROCESSOR HAVING DATA MEMORY SPACE MAPPED TO PROGRAM MEMORY SPACE WITH ERRONEOUS EXECUTION PROTECTION

FIELD OF THE INVENTION:

The present invention relates to a modified Harvard architecture processor having data memory space mapped to program memory space and having protection for erroneous execution of data entries in the program memory space.

BACKGROUND OF THE INVENTION:

Processors, including microprocessors, digital signal processors and microcontrollers, operate by running software programs that are embodied in one or more series of program instructions stored in a memory. The processors run the software by fetching the program instructions from the series of program instructions, decoding the program instructions and executing them. In addition to program instructions, data is also stored in memory that is accessible by the processor. Generally, the program instructions process data by accessing data in memory, modifying the data and storing the modified data into memory.

One well-known architecture for processors is known as the Harvard architecture. In this architecture, data and program instructions are stored in separate memories that can be accessed simultaneously. Because of this simultaneous access, the Harvard architecture provides significant processing speed advantages over other architectures. A typical Harvard architecture processor that includes internal memory includes two separate memories, one for data, and one for program instructions. In order to expand the memory

capacity of such a processor, memory external to the processor must be added. However, since a Harvard architecture processor has two separate memories, in order to expand both data memory and program instruction memory, two separate external memories must be added. This is a significant disadvantage when low-cost systems are being built.

A need arises for a processor having an architecture that provides the processing speed advantages of the Harvard architecture, but does not require two separate external memories in order to expand both data memory and program instruction memory. One solution to this problem is described in co-pending U.S. Patent Application No. XX/XXX,XXX. The described processor has separate program memory space and data memory space, but provides the capability to map at least a portion of the program memory space to the data memory space. This allows most program instructions that are processed to obtain the speed advantages of simultaneous program instruction and data access. It also allows program memory space and data memory space to be expanded externally to the processor using only one external memory device that includes both program instructions and data.

However, a problem arises with this solution. Under some circumstances, the processor may fetch and attempt to execute an entry in the program memory space that has been mapped to the data memory space and which contains data, not a program instruction. Such a situation may occur, for example, as a result of a bug in the software that is being executed. Attempted execution of data that is not a program instruction may cause unpredictable results. A need arises for a technique by which attempted execution of data that is not a program instruction may be detected and recovered from.

SUMMARY OF THE INVENTION:

The present invention is a method, and a processor implementing the method, that provides the capability to detect and recover from attempted execution of data that is not a program instruction in a processor in which at least a portion of a program memory space to a data memory space. This allows the processor to provide the speed advantages and expansion advantages without the risk of unpredictable program execution behavior.

According to the present invention, a method of operating a processor comprises the steps of: mapping at least a portion of a program memory space to a data memory space, storing an entry into the program memory space that is mapped to the data memory space, the entry comprising data and a protection opcode, fetching an entry from the program memory space, attempting to execute the fetched entry, trapping the protection opcode, vectoring to a trap handler, and executing the trap handler.

In one aspect of the present invention, the trap handler is an illegal instruction trap handler and the step of executing the trap handler comprises the steps of determining that the opcode is a protection opcode, and executing a software routine to handle the trap. The program memory space may be internal to the processor. The processor may be operably connected to an external memory device operable to store program instructions and data, the external memory device comprising program memory space.

In one aspect of the present invention, the trap handler is a protection trap handler. The program memory space may be internal to the processor. The processor may be

operably connected to an external memory device operable to store program instructions and data, the external memory device comprising program memory space.

BRIEF DESCRIPTION OF THE FIGURES:

The above described features and advantages of the present invention will be more fully appreciated with reference to the detailed description and appended figures in which:

Fig. 1 depicts a functional block diagram of an embodiment of a processor chip within which the present invention may find application.

Fig. 2 depicts a functional block diagram of a data busing scheme for use in a processor 100, such as that shown in Fig. 1.

Fig. 3 depicts an exemplary memory map of a data space memory, which may be implemented in the processor shown in Fig. 2.

Fig. 4 depicts an exemplary block diagram of program memory space to data memory space mapping which may be implemented in the processor shown in Fig. 2, according to the present invention.

Fig. 5 depicts a block diagram of a data execution protection scheme, which may be implemented in the processor shown in Fig. 2, according to the present invention.

Fig. 6 depicts a processing flow diagram of a process for detection and handling of erroneous execution of a data entry, which may be implemented in the processor shown in Fig. 2, according to the present invention.

DETAILED DESCRIPTION:

Overview of Processor Elements

Fig. 1 depicts a functional block diagram of an embodiment of a processor chip within which the present invention may find application. Referring to Fig. 1, a processor 100 is coupled to external devices/systems 140. The processor 100 may be any type of processor including, for example, a digital signal processor (DSP), a microprocessor, a microcontroller, or combinations thereof. The external devices 140 may be any type of systems or devices including input/output devices such as keyboards, displays, speakers, microphones, memory, or other systems which may or may not include processors. Moreover, the processor 100 and the external devices 140 may together comprise a stand alone system.

The processor 100 includes a program memory 105, an instruction fetch/decode unit 110, instruction execution units 115, data memory and registers 120, peripherals 125, data I/O 130, and a program counter and loop control unit 135. The bus 150, which may include one or more common buses, communicates data between the units as shown.

The program memory 105 stores software embodied in program instructions for execution by the processor 100. The program memory 105 may comprise any type of nonvolatile memory such as a read only memory (ROM), a programmable read only memory (PROM), an electrically programmable or an electrically programmable and erasable read only memory (EPROM or EEPROM) or flash memory. In addition, the program memory 105 may be supplemented with external nonvolatile memory 145 as shown to increase the complexity of software available to the processor 100.

Alternatively, the program memory may be volatile memory, which receives program instructions from, for example, an external non-volatile memory 145. When the program memory 105 is nonvolatile memory, the program memory may be programmed at the time of manufacturing the processor 100 or prior to or during implementation of the processor 100 within a system. In the latter scenario, the processor 100 may be programmed through a process called in-line serial programming.

The instruction fetch/decode unit 110 is coupled to the program memory 105, the instruction execution units 115, and the data memory 120. Coupled to the program memory 105 and the bus 150 is the program counter and loop control unit 135. The instruction fetch/decode unit 110 fetches the instructions from the program memory 105 specified by the address value contained in the program counter 135. The instruction fetch/decode unit 110 then decodes the fetched instructions and sends the decoded instructions to the appropriate execution unit 115. The instruction fetch/decode unit 110 may also send operand information including addresses of data to the data memory 120 and to functional elements that access the registers.

The program counter and loop control unit 135 includes a program counter register (not shown) which stores an address of the next instruction to be fetched. During normal instruction processing, the program counter register may be incremented to cause sequential instructions to be fetched. Alternatively, the program counter value may be altered by loading a new value into it via the bus 150. The new value may be derived based on decoding and executing a flow control instruction such as, for example, a branch instruction. In addition, the loop control portion of the program counter and loop control

unit 135 may be used to provide repeat instruction processing and repeat loop control as further described below.

The instruction execution units 115 receive the decoded instructions from the instruction fetch/decode unit 110 and thereafter execute the decoded instructions. As part of this process, the execution units may retrieve one or two operands via the bus 150 and store the result into a register or memory location within the data memory 120. The execution units may include an arithmetic logic unit (ALU) such as those typically found in a microcontroller. The execution units may also include a digital signal processing engine, a floating point processor, an integer processor, or any other convenient execution unit. A preferred embodiment of the execution units and their interaction with the bus 150, which may include one or more buses, is presented in more detail below with reference to Fig. 2.

The data memory and registers 120 are volatile memory and are used to store data used and generated by the execution units. The data memory 120 and program memory 105 are preferably separate memories for storing data and program instructions respectively. This format is a known generally as a Harvard architecture. It is noted, however, that according to the present invention, the architecture may be a Von-Neuman architecture or a modified Harvard architecture, which permits the use of some program space for data space. A dotted line is shown, for example, connecting the program memory 105 to the bus 150. This path may include logic for aligning data reads from program space such as, for example, during table reads from program space to data memory 120.

Referring again to Fig. 1, a plurality of peripherals 125 on the processor may be coupled to the bus 125. The peripherals may include, for example, analog to digital converters, timers, bus interfaces and protocols such as, for example, the controller area network (CAN) protocol or the Universal Serial Bus (USB) protocol and other peripherals. The peripherals exchange data over the bus 150 with the other units.

The data I/O unit 130 may include transceivers and other logic for interfacing with the external devices/systems 140. The data I/O unit 130 may further include functionality to permit in circuit serial programming of the Program memory through the data I/O unit 130.

Fig. 2 depicts a functional block diagram of a data busing scheme for use in a processor 100, such as that shown in Fig. 1, which has an integrated microcontroller arithmetic logic unit (ALU) 270 and a digital signal processing (DSP) engine 230. This configuration may be used to integrate DSP functionality to an existing microcontroller core. Referring to Fig. 2, the data memory 120 of Fig. 1 is implemented as two separate memories: an X-memory 210 and a Y-memory 220, each being respectively addressable by an X-address generator 250 and a Y-address generator 260. The X-address generator may also permit addressing the Y-memory space thus making the data space appear like a single contiguous memory space when addressed from the X address generator. The bus 150 may be implemented as two buses, one for each of the X and Y memory, to permit simultaneous fetching of data from the X and Y memories.

The W registers 240 are general purpose address and/or data registers. The DSP engine 230 is coupled to both the X and Y memory buses and to the W registers 240. The

DSP engine 230 may simultaneously fetch data from each the X and Y memory, execute instructions which operate on the simultaneously fetched data and write the result to an accumulator (not shown) and write a prior result to X or Y memory or to the W registers 240 within a single processor cycle.

In one embodiment, the ALU 270 may be coupled only to the X memory bus and may only fetch data from the X bus. However, the X and Y memories 210 and 220 may be addressed as a single memory space by the X address generator in order to make the data memory segregation transparent to the ALU 270. The memory locations within the X and Y memories may be addressed by values stored in the W registers 240.

Any processor clocking scheme may be implemented for fetching and executing instructions. A specific example follows, however, to illustrate an embodiment of the present invention. Each instruction cycle is comprised of four Q clock cycles Q1 – Q4. The four phase Q cycles provide timing signals to coordinate the decode, read, process data and write data portions of each instruction cycle.

According to one embodiment of the processor 100, the processor 100 concurrently performs two operations – it fetches the next instruction and executes the present instruction. Accordingly, the two processes occur simultaneously. The following sequence of events may comprise, for example, the fetch instruction cycle:

Q1: Fetch Instruction
Q2: Fetch Instruction
Q3: Fetch Instruction
Q4: Latch Instruction into prefetch register, Increment PC

The following sequence of events may comprise, for example, the execute instruction cycle for a single operand instruction:

Q1: latch instruction into IR, decode, and determine addresses of operand data
Q2: fetch operand
Q3: execute function specified by instruction and calculate destination address for data
Q4: write result to destination

The following sequence of events may comprise, for example, the execute instruction cycle for a dual operand instruction using a data pre-fetch mechanism. These instructions pre-fetch the dual operands simultaneously from the X and Y data memories and store them into registers specified in the instruction. They simultaneously allow instruction execution on the operands fetched during the previous cycle.

Q1: latch instruction into IR, decode, and determine addresses of operand data
Q2: pre-fetch operands into specified registers, execute operation in instruction
Q3: execute operation in instruction, calculate destination address for data
Q4: complete execution, write result to destination

An exemplary memory map of data space memory 300 is shown in Fig. 3. Data space memory 300 includes a plurality of blocks of memory, divided into X address memory and Y address memory. Typically, data space memory 300 is implemented as random access read-write memory (RAM), so as to allow data to be read and written as necessary. However, read-only memory (ROM) may also be advantageously used for at least a portion of data space memory 300. For example, constant data values, look up

tables, etc., may be usefully stored in ROM. In the example shown in Fig. 3, X address memory includes memory blocks 302, 304, 306, and 308, while Y address memory includes memory block 310. Data space memory 300 is split into two blocks, X address memory and Y address memory. A key element of this architecture is that the Y address memory space is a subset of the X address memory space, and is fully contained within the X address memory space. In order to provide an apparent linear addressing space, the X and Y address memory spaces would typically have contiguous addresses, although this is not an architectural necessity.

In the example shown in Fig. 3, memory block 302 includes a block of contiguous memory, starting at data memory location 0x0000. Memory block 302 is reserved in X address memory space and is directly addressable using memory direct instructions. The remaining X address memory and Y address memory spaces are indirectly addressable using other instructions.. In the example shown in Fig. 3, Y address memory space 310 is located between two blocks of X address memory space, block 304 and 306. However, this is only an example, as the Y address memory space 310 may be located anywhere within the non-reserved X address memory space. The partition between the X and Y address spaces is arbitrary and is determined by the memory decode shown in Fig. 2. Both the X and Y address generator can generate any effective address (EA) within the range of data memory space 300. However, accesses to memory addresses that are in the other address space, or to memory addresses that are not implemented with physical memory will return data of 0x0000 (all zeros).

Memory block 308 is shown in Fig. 3 as being an X address memory block. Memory block 308, which includes at least a portion of data memory space 300, may be used as X address memory, Y address memory, or a mixture of X address memory and Y address memory. When used as X address memory, memory block 308 may be mapped into program memory space. This provides transparent access to constant data, such as stored constants, look up tables, etc., from the X address data memory space without the need to use special instructions. This feature allows the mapping of a portion of data memory space into an unused area of program memory, and since all unused internal addresses are mapped externally, to the external memory bus. This is shown in Fig. 4, which is an exemplary block diagram of the program memory space to data memory space mapping. Data memory space block 308, which is a portion of data memory space 300 is mapped to a data memory space page 402 in internal program memory space 404. The location of data memory space page 402 in internal program memory space 404 is specified by page register 406. Internal program memory space 404 is still used for program instruction access, as specified by program counter (PC) 408.

External memory device 410 is connected to the external memory bus 412 of the processor. External memory device 410 includes external program/data memory space 414. Since all unused internal addresses are mapped externally to the external memory bus, data memory space mapped page 402 is also mapped to external data memory space mapped page 416, which is located in external program/data memory space 412. If external memory device 410 is a RAM, then data may be read from and written to external data memory space mapped page 416. External program/data space 414 may also include

external program memory space 418, which may be separate from external data memory space mapped page 416, or which may overlap with external data memory space mapped page 416.

Since the program memory space may include data that is used when a portion of the program memory space is mapped to the data memory space, there is some danger that the processor will erroneously fetch and attempt to execute that data. This may happen, for example, when there is a bug in a software program that is executing on the processor that sets the program counter (PC) to a memory location in the program memory space that happens to be storing data. This problem can arise when data is stored in internal program memory space and is even more likely to arise when data is stored in an external memory device. The present invention includes a mechanism for detecting such erroneous accesses and provides the capability to handle such errors.

A block diagram of the data execution protection scheme of the present invention is shown in Fig. 5. Data memory space 502, including a plurality of data entries 504, is mapped from a data memory block portion 506 of program memory space 508. Program memory space 508 also includes one or more blocks of program instructions, such as program instruction blocks 510 and 512. As shown, each data entry, such as data entry 514, in data memory space 502 includes 16 bits of data. Each program instruction entry, such as program instruction entry 516, in program memory space 508 includes 24 bits of program instruction. The entries in data memory block 506 of program memory space 508, such as entry 518, are likewise 24 bits. Since a data entry only requires 16 bits, such as data portion 520 of entry 518, 8 bits of each entry in data memory block 506 are not

used by data and may be used for other functions. In the present invention, this other portion is used to contain a protection opcode 522, which allows erroneous execution of a data entry to be detected.

A process 600 for detection and handling of erroneous execution of a data entry is shown in Fig. 6. The process begins with step 602, in which data is stored to the data memory space that was mapped from program memory space. This data is stored to the lower 16 bits of each entry that is used. In addition to the data that is stored, a protection opcode is stored to the upper 8 bits (byte) of each data entry that is used. Typically, the protection opcode will be stored when the data entry is stored. For example, since program memory is typically implemented using non-volatile memory, the program instructions stored in the program memory are stored to the program memory during the production process. The protection opcodes may easily be stored to the program memory by this step in the production process. This is true both for internal program memory and for non-volatile external memory.

In step 604, program memory space is mapped to data memory space by issuance of the proper program instructions. In step 606, the processor erroneously fetches and attempts to execute data that was stored in an entry in data memory space that was mapped from program memory space. Since the processor is fetching a program instruction, the processor treats the entry as a program instruction entry and fetches the entire 24 bits of the entry. The upper 8 bits of the entry are the protection opcode, while the lower 16 bits are the data in the entry. The processor attempts to execute the fetched entry, and in particular attempts to decode the protection opcode. In step 608, this attempted decode of the

protection opcode causes a processor trap to occur. A trap can be considered to be a non-maskable, nestable interrupt. They provide a means by which erroneous operation can be corrected during software debug and during operation of the software. Upon occurrence of a trap, the execution flow of the processor is vectored to a trap handler in step 610. That is, the program counter of the processor is loaded with a value that points to the trap handler. The trap handler is a software routine that takes the appropriate corrective action upon occurrence of the trapped condition. The value is stored in an exception vector table that includes vectors for a variety of exception conditions, such as reset, stack overflow, address error, illegal instruction trap, arithmetic error, etc. Each entry in the exception vector table points to an exception handler that takes the appropriate action upon occurrence of the corresponding exception. In step 612, the trap handler deals with the error. Typically, the trap handler simply forces a reset of the processor. This would be done, for example, in an implementation in which a stand-alone application is executing in the processor. Since an attempt to execute a data entry is likely a result of a serious program error, performing a reset of the processor is often the best way of recovering from such an error. In an implementation in which there is an operating system controlling the processor, it may be possible to simply terminate the application program that caused the error and allow the operating system to recover from the error.

In a preferred embodiment, the illegal instruction trap vector is used to vector the processor to a routine that handles the attempted execution of a protection opcode. The protection opcode must be one of the possible 8 bit opcodes that is not used by any instruction of the processor. Attempted execution of this opcode will result in an illegal

instruction trap. The illegal instruction trap handler must then examine the opcode that caused the illegal instruction trap, determine that the opcode is the protection opcode, and execute the appropriate software routines to handle the trap, which typically includes recovering from the error condition. Alternatively, there may be a defined protection trap that is separate from the illegal instruction trap. Attempted execution of the protection opcode will cause a protection trap to occur, rather than a general illegal instruction trap. Since the processor will have already determined the opcode that was attempted to be executed was the protection opcode, the protection trap then need only execute the appropriate software routines to handle the error condition.

In the embodiment described above, internal program memory is organized as a plurality of 24 bit entries, each of which may contain a 16 bit data entry and an 8 bit protection opcode. The present invention also contemplates a number of additional and alternative embodiments. For example, an external memory may be used in which 24 bit entries are stored. In this embodiment, a 24 bit entry may contain a 16 bit data entry and an 8 bit protection opcode. If the external memory is a non-volatile memory, then the data entries and protection opcodes, along with any program instructions, may be stored in the external memory during the production process. If the external memory is a volatile memory, then the data entries and the protection opcodes must be stored to the external memory by the processor.

Alternatively, data entries may be stored in the external memory as 16 bit data entries, without protection opcodes. In this embodiment, the external memory may be connected to the processor using a memory bus configuration that is aware that the data

entries are 16 bits. For example, the memory bus connected to the external memory may be 16 bits wide, rather than the 24 bits wide that would be needed for program instructions. As another example, the address range of the external memory that is mapped to data memory may be used by the processor to identify a portion of the external memory that is storing data entries rather than program entries. In either example, the processor can detect an attempted program instruction access of the external memory or the portion of external memory that is storing data entries. Upon detection of such an attempted access, the processor may directly perform a protection trap. Alternatively, the processor may simply force a protection opcode onto the top 8 bits of the program instruction bus, which will also cause a protection trap to be performed.

While specific embodiments of the present invention have been illustrated and described, it will be understood by those having ordinary skill in the art that changes may be made to those embodiments without departing from the spirit and scope of the invention. For example, the present invention has been described in terms of 16 bit data entries, 24 bit program instruction entries, and 8 bit opcodes. However, one of skill in the art will recognize that such specific values are only examples, and that other arrangements and numbers of bits may be used without departing from the spirit and scope of the invention. The present invention contemplates any and all such alternative arrangements and numbers of bits.